
```
for(num2 = 0; num2<=9; num2++)
{
  for(num1=0; num1<=9; num1++)
  {
    System.out.println(num2+ "  "+ num1);
  }
}
```

outer

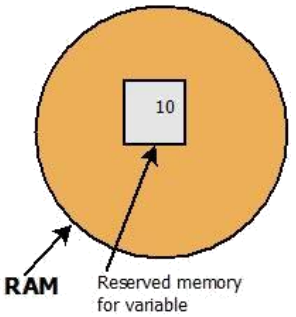
inner

```
int j=10;
```

Data Type

Variable Name

Stored Value in variable



Fundamental Data Types, Decision, and Iteration

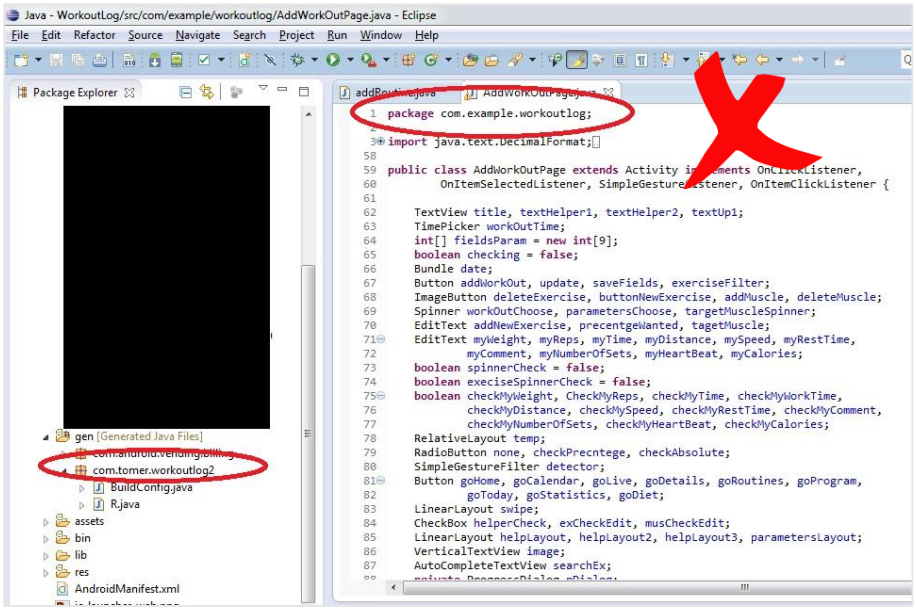
ITCS 209

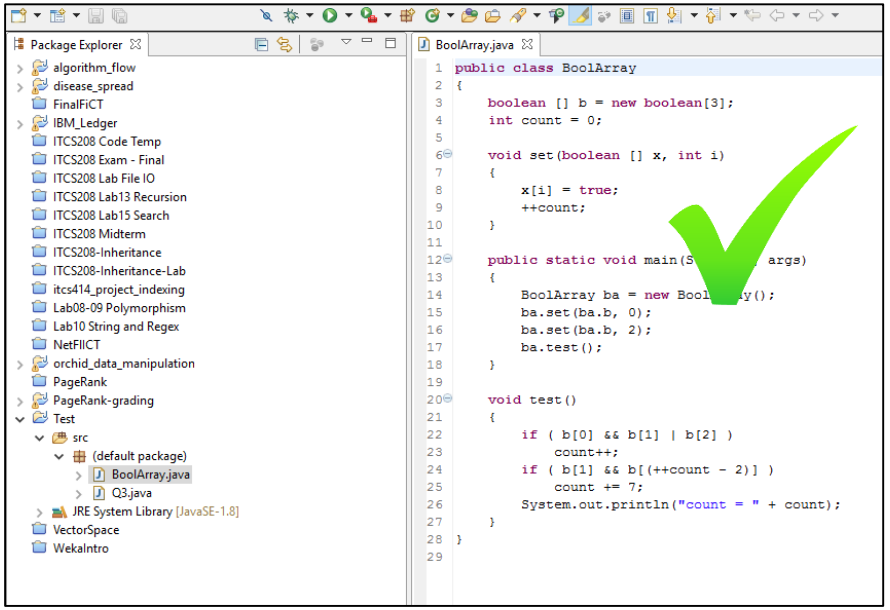
Assoc. Prof. Dr. Suppawong Tuarob

Faculty of Information and Communication Technology

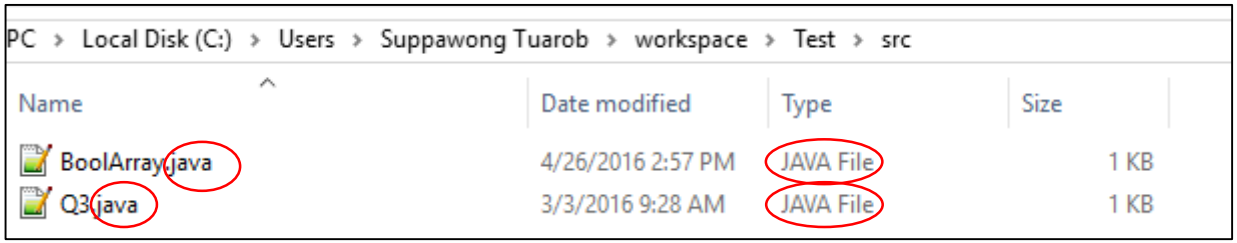
Common mistakes in last week's lab

- Do not use packages (yet)! Make sure your .java files are in "Default Package"



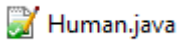


- Do not submit the entire project! Only .java files.

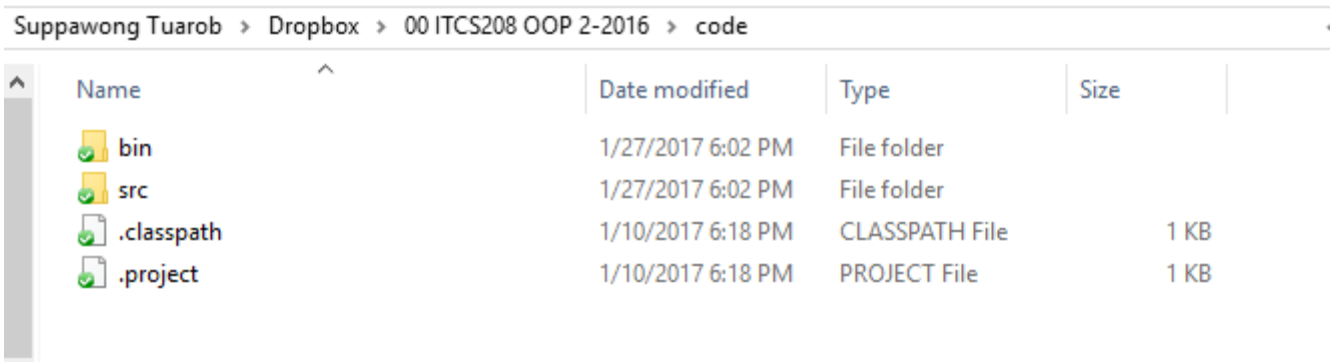


Importing an Existing Project to Eclipse

- Only .java files



- A project directory



Objectives

- To understand how to appropriately use data types
- To understand the proper use of constants
- To write arithmetic expressions in Java
- To learn how to read program input and produce formatted output
- To learn how to write a decision statements
- To learn how to write a iteration statements

Outline

- Data Types
- Constants
- Assignment and Comparison
- Arithmetic Operation
- Increment, Decrement
- Reading Input
- Decision
- Iteration

Data Types

Data types

- In Java, every variable (i.e. identifier) is either
  - a reference to an object

```
e.g. Car car1 = new Car();
     String text = "string is an obj";
```

How to notice:
1. Type starts with a capital letter: ***C***ar
2. Normal font style in Eclipse

  - or belongs to eight primitive types

```
e.g. int count = 3;
     boolean check = true;
```

How to notice:
1. Type starts with a lowercase letter: ***i***nt, ***b***oolean
2. **Purple-bold** font style in Eclipse

Stack and heap

Stack is used for static memory allocation. The "size" of each variable's value is fixed once created.

Heap is for dynamic memory allocation. The size of each object can increase/decrease dynamically.

```
int i = 9;
char c = 'K';
Dog d = new Dog("Smith");
```

Variable Stack

name	value
i	9
c	'K'
d	0x111
…	

Heap

0x110
"Smith"

When do object references behave differently from primitive variables

- Parameter passing (to a method)

```
public void doSomething(Dog d, int k){ … }
```

- Assignment

```
Dog d1 = new Dog();
Dog d2 = d1;       //d1 and d2 refer to the same object

int k1 = 5;
int k2 = k1;       //k1 and k2 have their own values (= 5)
```

- Quality (==)

```
Dog d1 = new Dog();
Dog d2 = new Dog();
if(d1 == d2){ //d1 refers to the same object as d2}
else { //d1 and d2 refer to different objects}

int a = 5; int b = 5;
if(a == b) {a stores the same value as b}
else {a stores different value from b}
```

To instantiate an object from a class

- Use new command

```
Dog d = new Dog();
StringBuilder str = new StringBuilder("Initial String");
```

- Exception for some *frequently used* immutable classes

```
String s = "Initial String";
    //(almost) same as String s = new String("Initial String");
Double d = 0.009;
Integer i = 5;
Character c = 'K';
```

null

- If you want to specify or initialize an object reference NOT to refer to any object.

```
Dog d = null;
//In C/C++: int* ptr = 0;
```

Eight primitive data types

Table 1 Primitive Types		
Type	Description	Size
`int`	The integer type, with range –2,147,483,648 (`Integer.MIN_VALUE`) . . . 2,147,483,647 (`Integer.MAX_VALUE`, about 2.14 billion)	4 bytes
`byte`	The type describing a single byte, with range –128 . . . 127	1 byte
`short`	The short integer type, with range –32,768 . . . 32,767	2 bytes
`long`	The long integer type, with range –9,223,372,036,854,775,808 . . . 9,223,372,036,854,775,807	8 bytes
`double`	The double-precision floating-point type, with a range of about $\pm 10^{308}$ and about 15 significant decimal digits	8 bytes
`float`	The single-precision floating-point type, with a range of about $\pm 10^{38}$ and about 7 significant decimal digits	4 bytes
`char`	The character type, representing code units in the Unicode encoding scheme (see Computing & Society 4.2 on page 163)	2 bytes
`boolean`	The type with the two truth values `false` and `true` (see Chapter 5)	1 bit

Eight primitive data types

- In practice, we will mostly use:

`boolean` to represent logic

`int`, `long` and `double` to represent numbers

`char` to represent a character

Converting floating-point to Integer (Type casting)

- In Java you cannot directly assign `double` to `int`

  e.g.
  ```
  double balance = total + tax;
  int dollars = balance;
  //Error: Cannot assign double to int
  //Why?
  //To prevent you from unintentionally losing precision
  ```

- To fix this, you should use the **cast** operator `(int)` to convert a floating-point to an integer

```
double balance = total + tax;
int dollars = (int) balance;
```

****In this case**: if balance is 13.75, then dollars is set to 13.

Converting floating-point to Integer

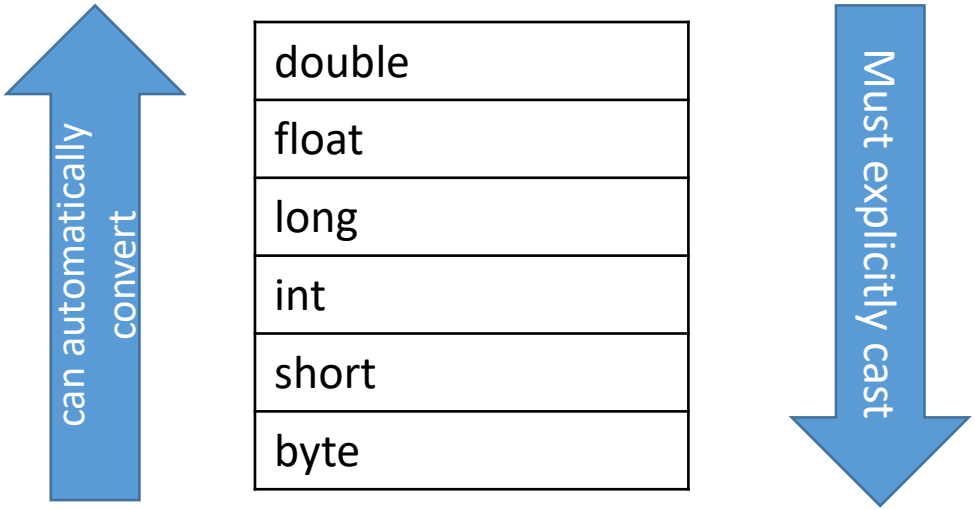
- Discarding the fractional part is not always appropriate.

- If you want to round a floating-point number, use the `Math.round` method. Be careful, this method returns value as a **long** integer

```
e.g., long rounded = Math.round(balance);
```

Therefore, if balance is 13.75, then round is set to 14.

More precise -> less precise (numbers)



Converting floating-point to Integer

- Example

```
double a = 126.254;
int result1 = (int) a;
```

?

```
double a = 126.254;
int result1 = Math.round(a);
```

?

double
float
long
int
short
byte

String

- Strings, though not of a primitive type, are frequently used in Java.
- ***Every*** class has method toString() implemented.
- Declaring String:

```
String str1 = "ABC";
    //only one String object is created in the heap

String str2 = new String("XYZ")
          //two objects of "XYC" are created in the heap. The
                first "XYC" is used to initialized the other
                "XYC" which is referenced by str2.
```

Strings

- There are many operations defined for String

Statement	Result	Comment
`string str = "Ja";` `str = str + "va";`	`str` is set to `"Java"`	When applied to strings, + denotes concatenation.
`System.out.println("Please"` `+ " enter your name: ");`	Prints `Please enter your name:`	Use concatenation to break up strings that don't fit into one line.
`team = 49 + "ers"`	`team` is set to `"49ers"`	Because `"ers"` is a string, 49 is converted to a string.

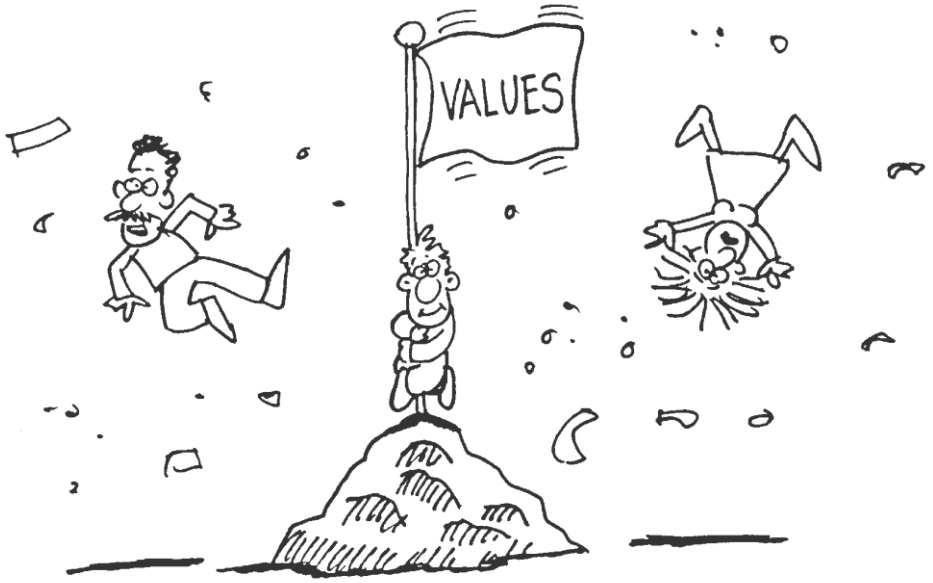
`String first = in.next();`
`String last = in.next();`
`(User input: Harry Morgan)`
	`first` contains `"Harry"`
	`last` contains `"Morgan"`
		The `next` method places the next word into the string variable.

`String greeting = "H & S";` `int n = greeting.length();`	`n` is set to 5	Each space counts as one character.

Strings (Cont)

Statement	Result	Comment
`String str = "Sally";` `char ch = str.charAt(1);`	`ch` is set to `'a'`	This is a `char` value, not a `String`. Note that the initial position is 0.
`String str = "Sally";` `String str2 = str.substring(1, 4);`	`str2` is set to `"all"`	Extracts the substring starting at position 1 and ending before position 4.
`String str = "Sally";` `String str2 = str.substring(1);`	`str2` is set to `"ally"`	If you omit the end position, all characters from the position until the end of the string are included.
`String str = "Sally";` `String str2 = str.substring(1, 2);`	`str2` is set to `"a"`	Extracts a `String` of length 1; contrast with `str.charAt(1)`.
`String last = str.substring(` `  str.length() - 1);`	`last` is set to the string containing the last character in `str`	The last character has position `str.length() - 1`.

Constant



Constants

- **Constant** is a variable that once its value has been set, it *cannot be changed*.

- Using constants makes your program easier to maintain and read.

  - Ex. Use `PI` instead of `3.14159265358979323846264338327950288419716939937510582097494459230781640628`

Usage: a keyword **`final`** is used in front of data type

E.g., `public static final double PI = 3.1415926535897932;`

Constant with Static

- Constants are typically safe to declare **static**
  - Their values can never be changed anyway.

- The **`static`** constants allow other classes to use them directly

```
public static final double PI = 3.14159265358979323;
```

Constant with Static (Cont)

- Example **Math** class

```
public class Math {
. . .
  public static final double E = 2.71828182845904523;
  public static final double PI = 3.14159265358979323;
}
```

- This constant can be referred in other class as:

```
  double circumference = Math.PI * diameter;
```

Enum


Days of the Week
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday

Enum in Java

- If you want to define your own variable type with specific set of possible values, you can use Enum
  - Days of Week
  - Gender
  - Color

```
public enum Color {Black, Blue, Green, Red };
```

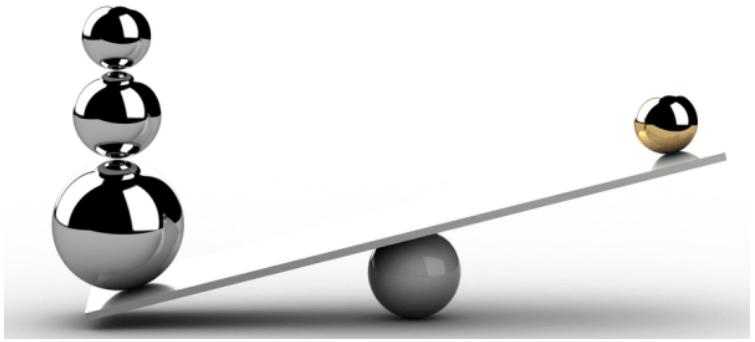
```
public class HelloOOP {

    public enum Color {Black, Blue, Green, Red };

    public static void main(String[] args)
    {
        Color c1 = Color.Black;

        Color c2 = Color.Yellow;
    }
}
```

Assignment and Comparison



Assignment

- **Assignment**: a statement to assign the result of an expression (on the right) to variable (on the left).

```
variable = expression;   ( e.g., int a = (b*c)/10; )
```

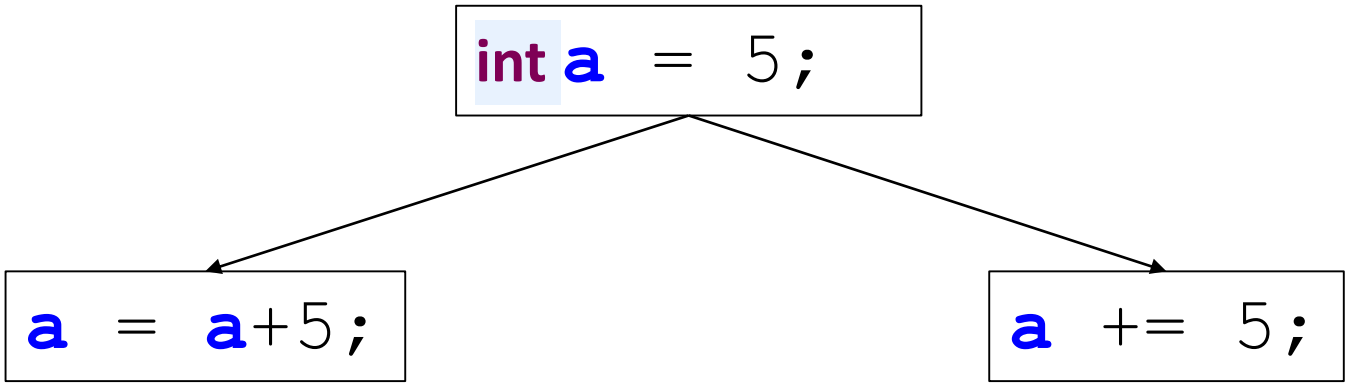
- In JAVA there is a compound assignment operator which is an operator that do both calculation and assignment at the same time:

Operator	Description
+=	Addition and assignment
-=	Subtraction and assignment
*=	Multiplication and assignment
/=	Division and assignment
%=	Remainder and assignment

Including ++, --

Compound assignment operator

- Example




```
Result -> a = 10;
```

Comparing values

- Relational operators are used to compare numbers.

Operator	Use	Example
<	Less than	`if(a<b)`
<=	Less than or equal to	`if(a<=b)`
>	Greater than	`if(a>b)`
>=	Greater than or equal to	`if(a>=b)`
==	Equal	`if(a==b)`
!=	Not equal	`if(a!=b)`

- Relational operations return ***boolean*** values (either `true` or `false`).

Comparing values (Cont)

- Be careful and Don't confuse them
  - The == operator tests for equality

```
if (x == 0) // if x equals zero
```

  - The = operator assigns a value to a variable

```
x = 0;  // x ← 0
```

Comparing Strings

- Don't use `==` for Strings!

```
if (input == "Y"){…} // WRONG!!!
```

In fact don't use == to compare the "states" of two **objects**

For two object references, == check whether they store the same **object addresses**

- Use `equals` method:

```
if (input.equals("Y")){…}
```

Comparing objects

- == is used for testing the actual value of attribute.
- The value of an object reference is the memory address of the referenced object.
- If you compare two object with the == operator, you can test whether the references refer to the same object.

Comparing values (Cont)

```
Rectangle box1 = new Rectangle(5, 10, 20, 30);
Rectangle box2 = box1;
Rectangle box3 = new Rectangle(5, 10, 20, 30);
```

```
box1 == box2;        // true
box1 == box3;        // false
box2 == box3;        // false
box1.equals(box3); // true
box2.equals(box3); // true
box1.equals(box2); // true
```

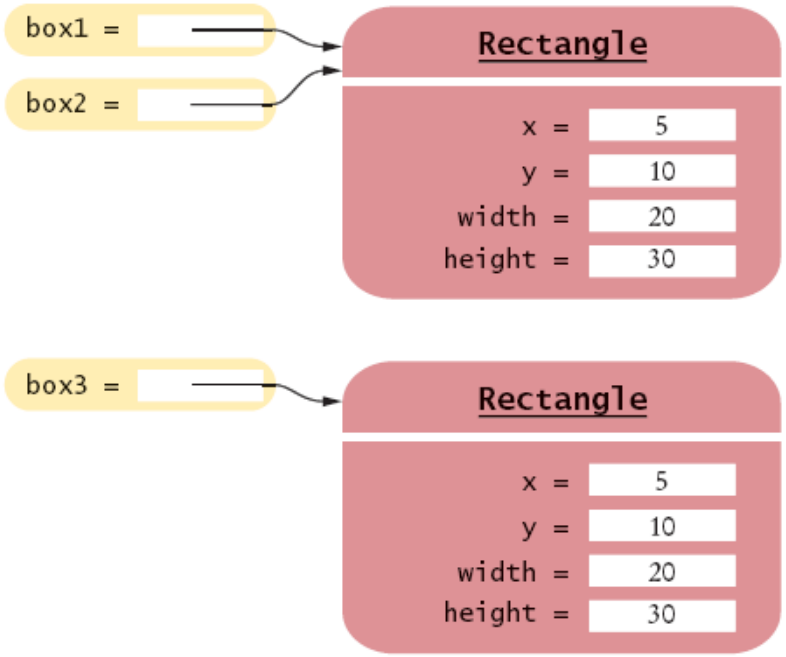


Figure 4 Comparing Object References

Testing for null

- An object reference can have the special value `null` if it refers to no object at all.

```
String middleInitial = null; // Not set
if(. . .){

 middleInitial = middleName.substring(0, 1)
}
```

- Note that the **null reference** is not the same as the empty string "". The empty string is a valid string object of length 0, whereas a null indicates that a string variable refers to no string at all.

Logical Operations

- **Boolean expression:** a logical statement that either true or false.
- You can combine boolean expressions with logical operators.

Operand	Example	Meaning
&&	var1 && var2	Are both values true?
||	var1 || var2	Is at least one value true?
!	!var1	Is it NOT var1?

Logical Operations

- **Example**

```
If( a==b || c < d){
      \\Statements
}
```

A	B	A AND B	A OR B	NOT A
False	False	False	False	True
False	True	False	True	True
True	False	False	True	False
True	True	True	True	False

Using Boolean Expressions

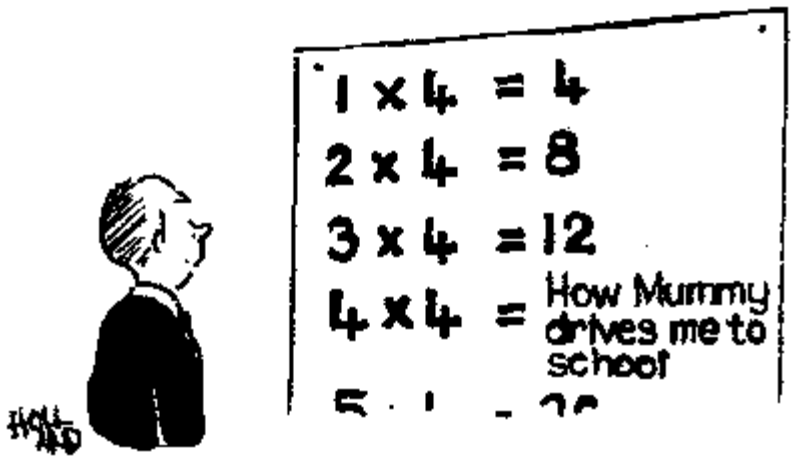
Practice…

“Suppose you write a program to show status of an air-conditioner that it is working properly when machine temperature is in range 15 to 40 Celsius”

```
if (condition)
        { System.out.println(“is working"); }
```

?

Arithmetic Operation



Arithmetic Operations

$$a = \underbrace{b + c * d}_{\textbf{Expression}}$$

(variable + operand)

Operator
Operand

Arithmetic Operations (cont)

Operator	Precedence
()	High
Unary Operator (-)	
*, /, %	
+, -	Low

Arithmetic Operations (Cont)

$$2 * (-5 - (17\%3/2) + 26) * 2 + 4 \quad = ?$$

Increment and Decrement

++ --

Increment and Decrement

- Increment and decrement operator are **Unary operator**
- Unary Operator Operates on **One Operand.**

- **Increment Operator (++)** is used to Increment Value Stored inside Variable on which it is operating.

  E.g., num++, ++num (**post** and **pre** increment)

- **Decrement Operator (--)** is used to decrement value of Variable by 1 (default).

  E.g., num--, --num (**post** and **pre** decrement)

Increment and Decrement

Practice

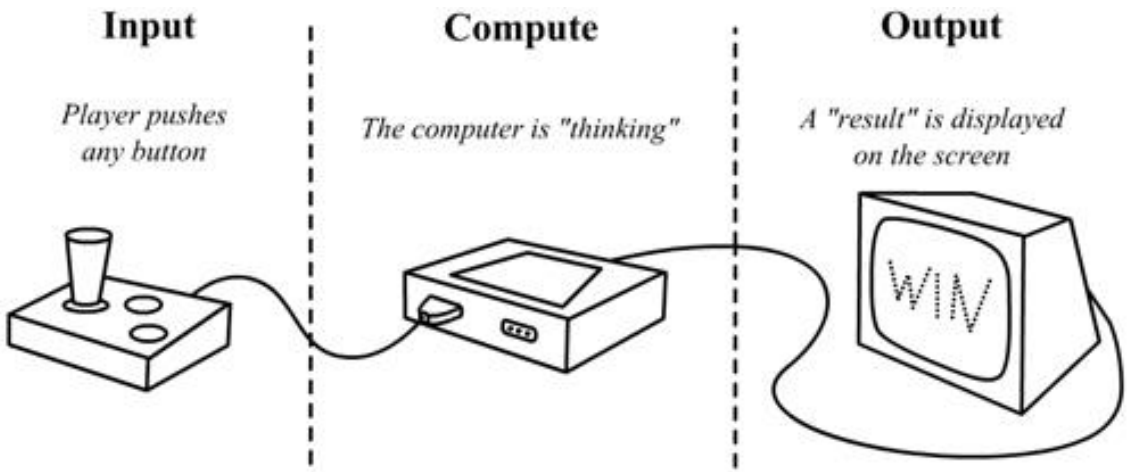
```
int i = 10;
int j = 10;

i++;
j++;

    System.out.println("i=" + i);
    System.out.println("j=" + j);

int k = i++;
int l = ++j;
    System.out.println("k=" + k);
    System.out.println("l=" + l);
```

Reading Input



Basic Input and Output

- You can make a program more flexible if you ask user for inputs rather than fixed values

E.g.

```
System.out.print("Please enter the number of bottles: "); //
Display prompt
```

Basic Input and Output

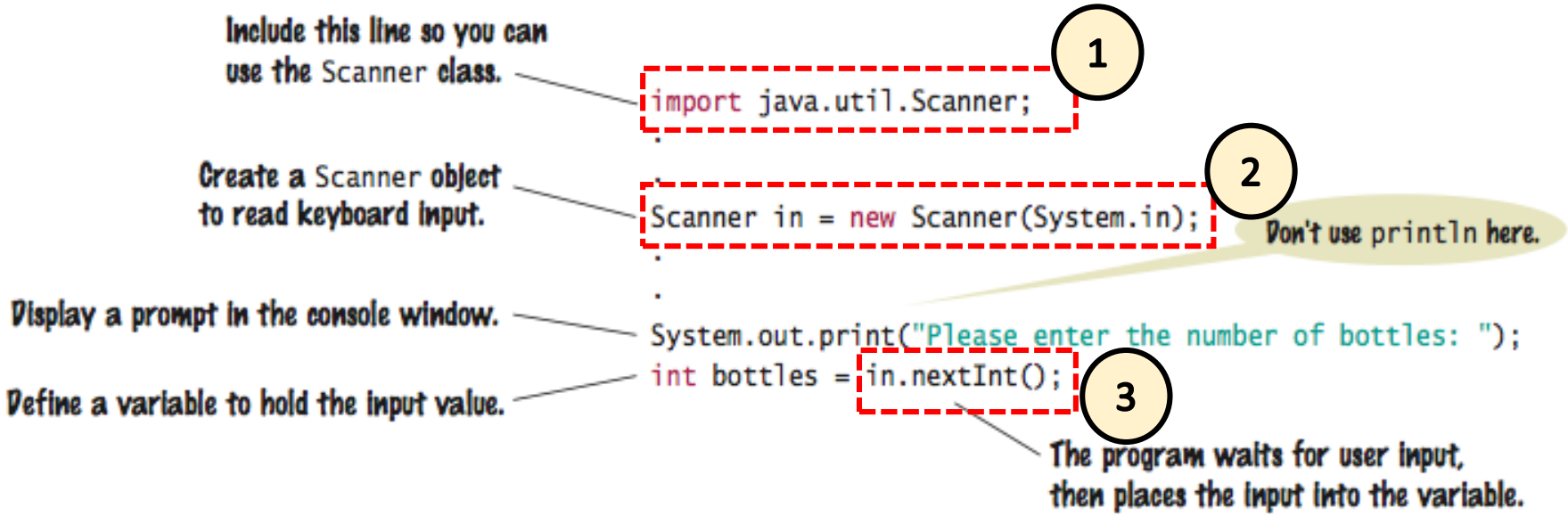
- Using the `Scanner` class to read keyboard input in a console window

```
Scanner in = new Scanner(System.in);
```

- Once you have a scanner, you use its `nextInt` method to read an integer value:

```
System.out.print("Please enter the number of bottles: ");
int bottles = in.nextInt();
```

Basic Input and Output



Basic Input and Output

- To read a floating-point number, use the `nextDouble` method:

```
System.out.print("Enter price: ");
double price = in.nextDouble();
```

- To read a String, use the `nextLine` method instead:

```
System.out.print("Enter a sentence: ");
double price = in.nextLine();
```

Decision

Decision

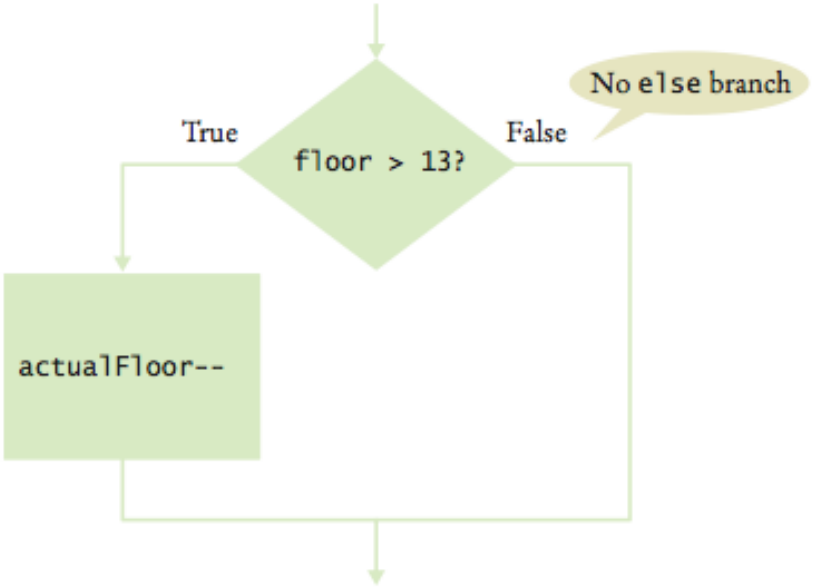
- If/else statement
- Multiple Alternatives
- Switch Statement
- Comparing values
- Boolean Expressions

if statement

- The if-then statement is the most basic of all the control flow statement. It tell your program to execute a certain section of code *only* if a particular test evaluates to *true*.

```
int actualFloor;

if (floor > 13)
{
   actualFloor = floor - 1;
}
```

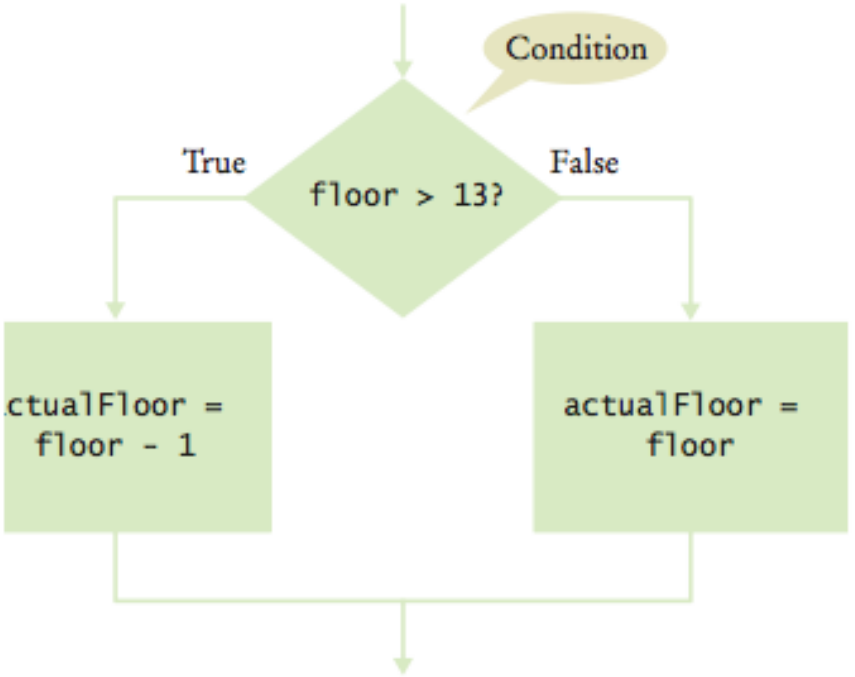


If-then-else statement

- The if-then-else statement provides a secondary path of execution when an "*if*" clause evaluates to *false*.

```
int actualFloor;

if (floor > 13)
{
   actualFloor = floor - 1;
}
else
{
   actualFloor = floor;
}
```



Multiple alternative

- The common use for multiple alternative is if-else-if

```
int actualFloor;

if (floor > 13){

  actualFloor = floor - 1;

}
else if (floor == 12){

  System.out.println("end of the actual floor");
  actualFloor = floor;

}else{

  actualFloor = floor;

}
```

Switch Statement

- **Switch**: also use to create multiple alternative by allowing a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

```
switch(expression) {
   case value :
   // Statements
   break; // optional
   case value :
   // Statements
    break; // optional
   // You can have any number of case statements.
   default : // Optional
   // Statements
}
```

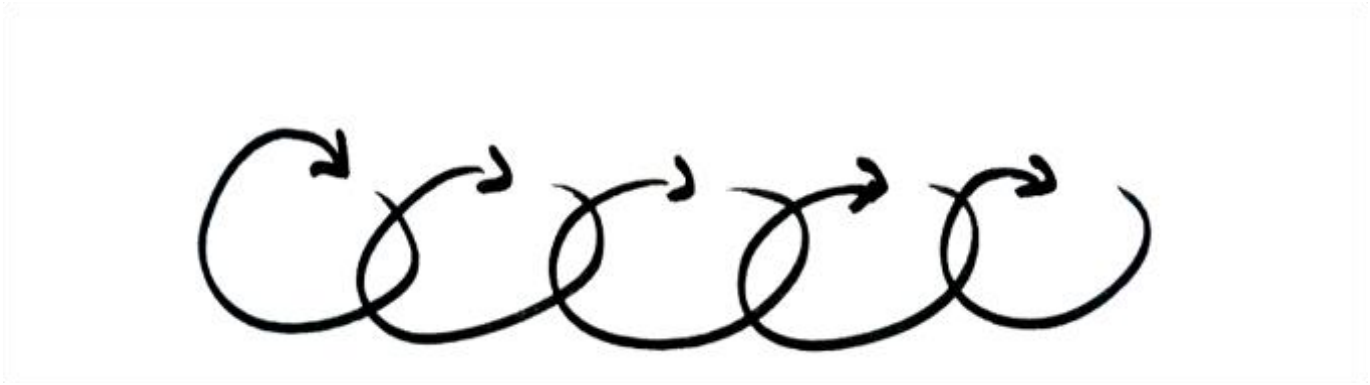
Switch Statement (Cont)

- **Switch:** also use to create multiple alternative by allowing a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

```
int digit = . . .;
if (digit == 1)       { digitName = "one"; }
else if (digit == 2) { digitName = "two"; }
else if (digit == 3) { digitName = "three"; }
else if (digit == 4) { digitName = "four"; }
else if (digit == 5) { digitName = "five"; }
else if (digit == 6) { digitName = "six"; }
else if (digit == 7) { digitName = "seven"; }
else if (digit == 8) { digitName = "eight"; }
else if (digit == 9) { digitName = "nine"; }
else { digitName = ""; }
```

```
int digit = . . .;
switch (digit)
{
case 1: digitName = "one"; break;
case 2: digitName = "two"; break;
case 3: digitName = "three"; break;
case 4: digitName = "four"; break;
case 5: digitName = "five"; break;
case 6: digitName = "six"; break;
case 7: digitName = "seven"; break;
case 8: digitName = "eight"; break;
case 9: digitName = "nine"; break;
default: digitName = ""; break;
}
```

Iteration



Iteration

- While Loop
- For Loop
- Do Loop

While Loop

- **While loop**: a statement for repeatedly executes a body statement as long as a given condition is true

```
While(condition){
 //Statements
}
```

While Loop

Practice 1...

```
int i = 0;
int sum = 0;
while (sum < 10)
{
   i++;
   sum = sum + i;
   System.out.println(i+","+sum);
}
```

While Loop

Practice 2...

```
int i = 0;
int sum = 0;
while (sum < 10)
{
   i++;
   sum = sum - i;
   System.out.println(i+","+sum);
}
```

For Loop

- **For Loop:** an execution of a body statements in a specific number of times.
- It is useful when you know how many times a task is to be repeated.

```
for(initialization,; condition; update){
  //Statements
}
```

For Loop

Practice ...

```
int i;
for (i=0; i<=5; i++)
{
   System.out.println(i);
}
```

For Loop

Practice ...

```
int i;
for (i=8; i>=0; i--)
{
   System.out.println(i);
}
```

For Loop

Practice ...

```
int i;
for (i=0; i<=9; i=i+3)
{
   System.out.println(i);
}
```

Do…while Loop

- **Do… while Loop**: this is similar to while loop apart from that this loop is guaranteed to execute at least one time.

```
do{
    //Statements
}while(condition);
```

Do...while Loop

Practice ...

```
int i = 0; int sum = 9;
do{
    i++; sum = sum + i;
    System.out.println(i+","+sum);
} while (sum < 10)
```

Do...while Loop

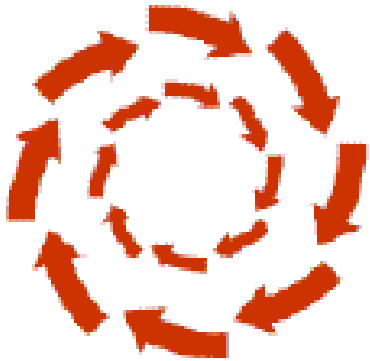
Practice ...

```
int i = 0; int sum = 11;
do{
    i++; sum = sum - i;
    System.out.println(i+","+sum);
} while (sum < 10)
```

Nested Loop

- **Nested Loop:** a placing of one loop inside the body of another loop is called **nesting.**

```
For(initialisation,; Boolean_expression; update){
   For(initialisation2,; Boolean_expression2; update2){
     //Statements
   }
}
```



Nested Loop (Cont)

- **Example1:** webpage counter

0	0

Considering 2 digits webpage counter...

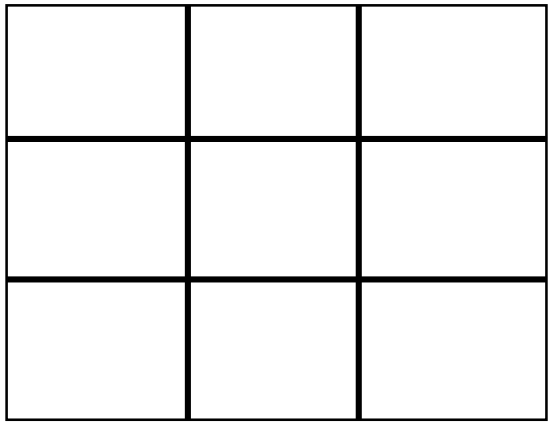
```
for(num2 = 0; num2<=9; num2++)
{
  for(num1=0; num1<=9; num1++)
  {
    System.out.println(num2+ "  "+ num1);
  }
}
```

outer

inner

Nested Loop (Cont)

- **Example2:** Table position



```
For(int row=0; row<3; row++){
   For(int col=0; col<3; col++){
     System.out.println("table position="+row+","+col);
   }
}
```

For Loop

Practice …

```
for (int i=1; i<=3; i++)
{
   for (int j=1; j<=4; j++)
   {
       System.out.print("*");
   }
    System.out.println();
}
```

For Loop

Practice …

```
for (int i=1; i<=4; i++)
{
   for (int j=1; j<=i; j++)
   {
       System.out.print("*");
   }
    System.out.println();
}
```

Let's move to the lab…